Lab 5 - MirrorArt

Overview

For this lab, our purpose is to practice using development strategies to write complex programs. We will use problem-solving techniques such as decomposition, “table technique”, multipliers and constants (i.e. basic algebra), and class constants to write our program.

Our development strategy is to build our program in stages. Instead of building the whole program all at once, we will build the program in small pieces. Once we finish with one small piece, we work on the next small piece. When we are all done building the small pieces, eventually our whole program will be finished. This strategy is called iterative enhancement (i.e. stepwise refinement).

Problem 1

Create a program that produces the following output on the console:

```
#================#
|       <><>     |
|     <>....<>   |
|   <>........<> |
|<>..............<>|
|<>..............<>|
|   <>........<> |
|     <>....<>   |
|       <><>     |
#================#
```

1.1 Create a new Java file called `MirrorArt`.

1.2 Decompose the required output into smaller problems. Recall Exam 1, problems 9-12, in which we had to implement the methods for the output:

```
  ______
 /......\
/........\
\......../
 \______/

\......../
 \______/
+--------+


  ______
 /......\
/........\
|  STOP  |
\......../
```

```
 \______/


  ______
 /......\
/........\
+--------+
```

And here's the source code to print out that output:

```
public class Figure {
   public static void main(String[] args) {
      topHalf();
      bottomHalf();
      System.out.println();
      bottomHalf();
      someLine();
      System.out.println();
      topHalf();
      stop();
      bottomHalf();
      System.out.println();
      topHalf();
      someLine();
   }
```

For `MirrorArt`, write a main method that decomposes the required output into smaller pieces that correspond to static methods. (hint: the required output suggests at least three static methods).

1.3 Create "stubs" for the static methods you used in 1.2. "Stubs" are simply static methods with no code in it. For example:

```
public static void main(String[] args) {
    doSomething();
}

public static void doSomething() {
    // TODO: add code
}
```

Use this same pattern above and write your stubbed methods. Compile your program to ensure that it is sytacticially correct. If compilation fails, look at the error logs, fix the errors, and then recompile until compilation passes. You should see nothing printed on the console because no code is in your stubbed methods. That is ok. Our goal is to get a high-level overview of the program by decomposing it into smaller problems.

1.4

- For the method that prints out `#================#`, write pseudocode (i.e. code written in plain English).
- Translate your pseudocode into Java code and put it in your corresponding method
- Compile your program to ensure that it is sytacticially correct. If compilation fails, look at the error logs, fix the errors, and then recompile until compilation passes.

1.5

- For the method that prints out:

```
|      <><>      |
|    <>....<>    |
|  <>........<>  |
|<>............<>|
```

- Write pseudocode that will produce the output above. To get you started, here's some starter pseudocode. Fill in the missing pieces (as denoted by `__`) of the pseudocode:

```
for (each of __ lines)
  print __
  print spaces (possibly zero)
  print __
  print __
  print <>
  print __
  print __
  println
```

- Translate as much of the pseudocode into real Java code. To get you startd, here's some starter Java code. Again, fill in the missing pieces (as denoted by `__`). Ignore the `??` for now. We will focus on determining `??` in the next task:

```
for (int line; line<= __; line__) {

  __

  for (int spaces = __; spaces <= ??; __) {

  }

  __

  for (int __ = __; __ <= ??; __) {

  }

  __

  for (int __ = __; __ <= ??; __) {

  }

  System.out.print(__);
  System.out.println();
}
```

- The hardest part of this small problem is determining `??`. What is `??` ? It is the 'test' section of the "for loop". How can we determine what the test shoudl be? Let's use the "table technique". We are going to

model the number of spaces and dots we need to print out per line on a table. Fill in the rest of the empty boxes.

line	spaces	dots
1	6	0
2	4	4
3		
4		

- For the number of spaces per line, we need to write some expression using the table above to determine how many spaces we need.
  - Recall from algebra, the slope intercept formula, `y = mx + b`. We will use this same idea to develop our expression to find the number of spaces per line.
  - In our problem, we need a multplier and a constant to help us determine the number of spaces per line.
  - Note how 'line' increases by one but 'spaces' decreases by two. How do we get from something that goes up from one to something that goes down by two? Answer: you multiply it by -2. Thus we get our multiplier: `(-2 * line)` But are we done yet? No. We need to add some constant to get to our desired answer. Well, what happens when `line = 1`? We get `(-2 * 1) = -2` But we don't want -2 spaces, we want 6 spaces. What do I add to -2 to get 6? Answer: we add 8 (ie. `(-2 * 1) + 8 = 6`) We can then generalize our expression for each line to `(-2 * line) + 8` Now, where should we plug in this expression in our code? Hint: we should use it to help us print out the number of spaces we need.
  - Use the table technique again but this time to write the expression to get the number of dots.
- Update your Java code with the expressions you determined in the previous two bullets. Hint: use those expressions to fill in the `??` of your Java code.
- Compile your program to ensure that it is sytacticially correct. If compilation fails, look at the error logs, fix the errors, and then recompile until compilation passes. When you run your program, it should produce the following:

```
#===============#
|       <><>      |
|     <>....<>    |
|   <>........<>  |
|<>............<>|
#===============#
```

When finished attach your `MirrorArt.java` file here.

1.6
(OPTIONAL/EXTRA CREDIT: When you have completed Problem 2, you have the option to complete this step to get extra credit (+3)):

For the method that prints out:

```
|<>.........<>|
|  <>.....<>  |
|    <>...<>    |
|      <><>      |
```

Implement the code for this output. Note how it looks very similar to some output in 1.5 but upside down. Can you reuse your code in 1.5 and modify it to create the desired output? You can also go through the same development process that we did in 1.5 and use it to develop the method for this output.